学习笔记

9.1 异常检测算法概要

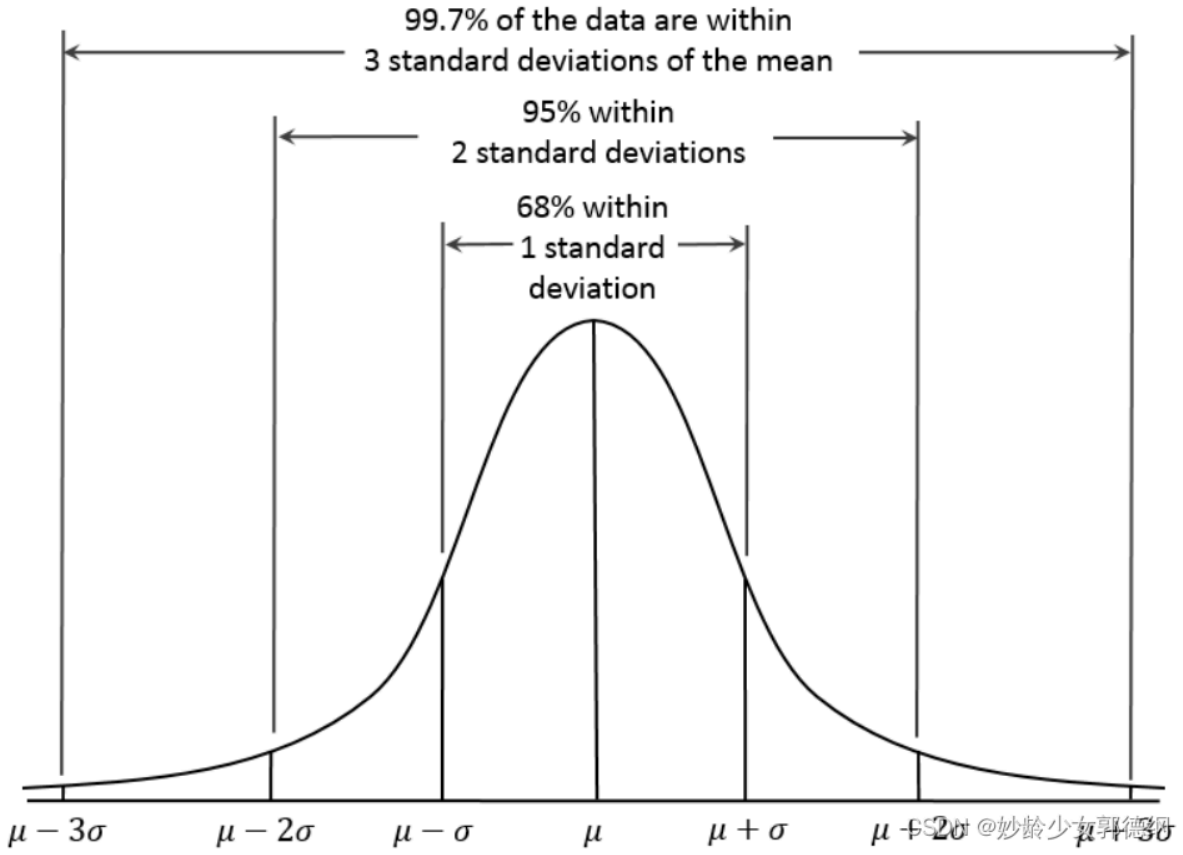
9.1.1 异常与异常检测

我对异常检测这一章节进行重点关注的原因是因为我自己的研究方向就是电力负荷预测，而在电力负荷预测研究中需要不断提升数据预测精度，这对数据处理水平要求极高，其中很重要的一个点就是数据异常点识别和处理，因此各类异常检测算法和原理都是需要我去深入理解和应用的，在阅读了这一章节的内容后，结合本学期所上的大数据分析课程的数据分析思路，我了解到数据异常是指与常规模式或预期行为不一致的数据点或活动。在不同领域，异常的定义和性质可能有所不同，如偏离点、变化点、奇异模式等。异常检测旨在从大量数据中识别出不寻常的行为或数据点，帮助发现潜在问题或安全威胁；主要难点包括数据的稀疏性、噪声与异常值的区分以及异常现象的解释。同时异常检测根据数据标签的可用性可分为监督学习、半监督学习和无监督学习；由于异常事件不常见且形式多变，异常检测多采用无监督学习方法。

在本章的第一小节中主要介绍了异常检测的定义、常见形式、主要难点和算法分类，基于此，我去CSDN上寻找了一些有关于异常检测的入门级讲解，并且还寻找了一些常用的异常检测算法以便后续学习应用。

【异常检测】数据挖掘领域常用异常检测算法总结以及原理解析（一） cof算法-CSDN博客

例如这一篇CSDN推文详细介绍了在数据分析领域常用的异常检测算法cof算法及其实现原理，这让我对异常检测的快速 入门有很大的帮助，首先就是异常检测的3sigma准则，这个准则在之前本科的微积分课程上就初步了解过，没想到在研究生阶段派上用场。



随后我还找了一些异常检测领域的应用型代码，以便后续学习过程中结合本书中自带的算法案例进行深入学习，其中有一篇CSDN推文的异常检测算法代码解析十分通俗易懂，文章链接如下：

一文详解8种异常检测算法（附Python代码）-CSDN博客

其中自编码器是一种强大的神经网络架构，主要用于告诉程序什么是正常数据，怎么寻找异常数据，对于异常检测算法的有效性来说至关重要，这篇文章介绍算法的自编码器代码示例如下所示。

```
# 构建自编码器（Autoencoder）网络模型
# 隐藏层节点数分别为16，8，8，16
# epoch为5，batch size为32

from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers

input_dim = X_train.shape[1]  # 输入数据的维度
encoding_dim = 16  # 编码层的维度
num_epoch = 5  # 训练周期
batch_size = 32  # 批量大小

# 构建自编码器模型
input_layer = Input(shape=(input_dim,))  # 输入层
encoder = Dense(encoding_dim, activation="tanh",
activity_regularizer=regularizers.l1(10e-5))(input_layer)  # 编码层1
encoder = Dense(int(encoding_dim / 2), activation="relu")(encoder)  # 编码层2
decoder = Dense(int(encoding_dim / 2), activation='tanh')(encoder)  # 解码层1
decoder = Dense(input_dim, activation='relu')(decoder)  # 解码层2

autoencoder = Model(inputs=input_layer, outputs=decoder)  # 创建自编码器模型

# 编译模型
autoencoder.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])
```

说明：

1. **模型构建**：
   - `input_dim`：输入数据的维度。
   - `encoding_dim`：编码层的维度，这里设置为16。
   - `num_epoch`：训练周期，设置为5。
   - `batch_size`：批量大小，设置为32。
2. **自编码器结构**：
   - 输入层：接收输入数据。
   - 编码层：通过两个`Dense`层逐步减少数据维度。
   - 解码层：通过两个`Dense`层逐步恢复数据维度。
3. **编译模型**：使用Adam优化器和均方误差损失函数，同时计算平均绝对误差（MAE）作为评估指标。

在后续学习中，可能会应用到这段自编码器的构造思路和流程。

9.1.2 异常检测算法的分类

随后，书中介绍了异常检测算法的分类，我了解到了异常检测算法根据作用原理和适用范围可分为四类：基于统计理论的算法、基于空间分布的算法、基于降维的算法和基于预测的算法。每种算法都有其独特的原理和应用场景，例如基于统计理论的算法利用数据的统计特性来识别异常值，而基于空间分布的算法则关注数据点在特征空间中的分布情况，其中，我在本学期所学习的大数据分析课程论文中就应用到了利用数据的统计特性来识别异常值的异常检测算法，其中的异常检测代码部分如下所示：

```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.ensemble import IsolationForest

def anomaly_detection():
    # 对数据进行1小时频率的重采样，并计算平均值
    df_resampled = df.resample('1h').mean()

    # 遍历每个国家
    for country in countries:
        # 获取当前国家的实际负荷数据，并将其重塑为二维数组
        data = df_resampled[f'{country}_actual'].values.reshape(-1, 1)

        # 初始化孤立森林模型，设置异常值比例为5%
        iso_forest = IsolationForest(contamination=0.05, random_state=42)
        yhat = iso_forest.fit_predict(data)

        # 获取最近30天的数据和对应的异常检测结果
        recent_data = df_resampled.iloc[-24*30:]
        recent_anomalies = yhat[-24*30:]

        # 计算最近30天的异常点数量
        anomaly_count = sum(recent_anomalies == -1)
        print(f"\n{country}最近30天异常点数量: {anomaly_count} (占比:
{(anomaly_count/(24*30)*100):.2f}%)")

        # 创建两个子图
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(15, 10))

        # 绘制完整时间序列的电力负荷和异常值
        ax1.plot(df_resampled.index, df_resampled[f'{country}_actual'],
                 label='实际负荷', alpha=0.6)
        ax1.scatter(df_resampled.index[yhat == -1],
                    df_resampled[f'{country}_actual'][yhat == -1],
                    color='red', label='异常值', s=20)
        ax1.set_title(f'{country}电力负荷异常值检测（完整时间序列）', fontsize=12,
pad=20)
        ax1.legend()

        # 绘制最近30天的电力负荷和异常值
        ax2.plot(recent_data.index, recent_data[f'{country}_actual'],
                 label='实际负荷', alpha=0.6)
        ax2.scatter(recent_data.index[recent_anomalies == -1],
                    recent_data[f'{country}_actual'][recent_anomalies == -1],
                    color='red', label='异常值', s=30)
        ax2.set_title(f'{country}电力负荷异常值检测（最近30天）', fontsize=12, pad=20)
        ax2.legend()
```

```
        # 调整布局并显示图表
        plt.tight_layout()
        plt.show()

# 示例数据和国家列表
# df = pd.read_csv('your_data.csv')  # 加载数据
# countries = ['Country1', 'Country2', 'Country3']  # 国家列表
# anomaly_detection()  # 调用函数
```

这段代码的运行结果会在后续文章中进行展示，这仅仅展示我所使用的异常检测思路。

9.1.3 异常检测的常用数据集

在第九章的这一小节中介绍了时间序列异常检测、图像异常检测和视频异常检测的常用数据集或者说是不同领域的异常检测算法常用的数据格式或者样式。例如，SMD数据集用于服务器机器数据的异常检测，MVTec AD数据集广泛应用于工业检测领域的异常检测。这些数据集为异常检测算法的评估和验证提供了重要的资源，在阅读了这一小节的内容之后，我了解到了在进行数据检测之前，需要先对数据进行预处理以便于程序能更高效的识别和处理异常数据点。

9.2 基于统计理论的异常检测

9.2.1 3σ准则

在这一小节中介绍了识别数据异常所使用的3σ准则，这个准则在前文中我自己所寻找的CSDN链接中就已经初步了解过了，只不过在书中本小节继续进行系统深入的学习。3σ准则（也称为拉依达准则）假设数据服从正态分布，这个准则通过计算数据点与均值的偏差来识别异常值，书中所展示在程序中使用该准则进行异常数据识别的代码示例如下所示：

```
# 示例代码：使用3σ准则检测异常值
import numpy as np

data = np.array([1, 2, 3, 4, 5, 100])  # 示例数据
mean = np.mean(data)
std_dev = np.std(data)

lower_bound = mean - 3 * std_dev
upper_bound = mean + 3 * std_dev

anomalies = [x for x in data if x < lower_bound or x > upper_bound]
print("异常值：", anomalies)
```

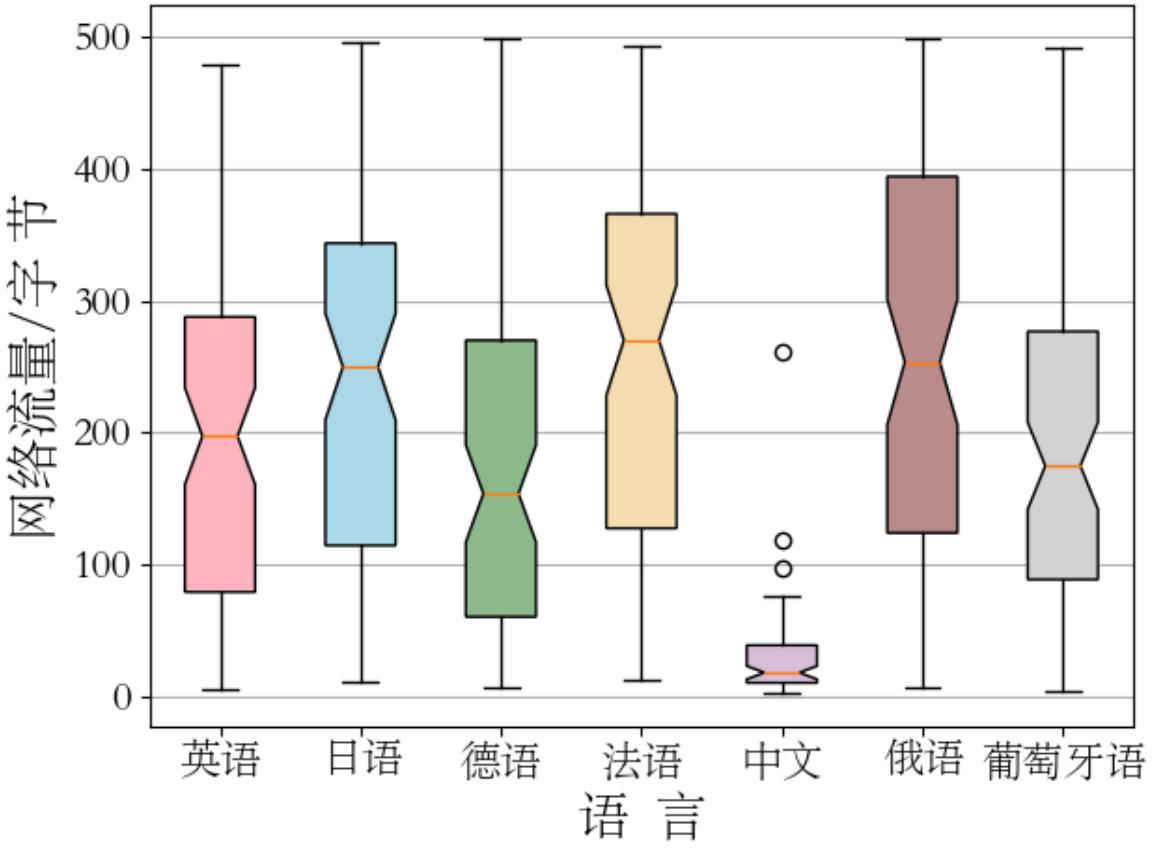
书中的这段代码让我对学会准则的python程序实现，并且可以基于此进行下一步的学习。

9.2.2 箱线图

通过后续阅读，我了解到了一种基础的一维特征空间非参数异常检测方法——箱线图，这个方法通过计算四分位距（IQR）来识别异常值。它不需要对数据进行特定的分布假设，适用于处理不符合正态分布的数据，书中代码示例如下所示。

```
import matplotlib.pyplot as plt

data = [1, 2, 3, 4, 5, 100]  # 示例数据
plt.boxplot(data)
plt.title("箱线图异常检测")
plt.show()
```

我通过查找相关资料和数据集，对这段代码进行应用，运行结果展示如下图所示：



9.2.3 基于直方图的异常值得分

基于直方图的异常值得分（HBOS）是一种多变量时间序列异常检测方法，该方法通过计算每个区间中数据点的密度来识别异常值。书中代码示例如下：

```
# 示例代码：基于直方图的异常值得分
import numpy as np

data = np.array([1, 2, 3, 4, 5, 100])  # 示例数据
hist, bin_edges = np.histogram(data, bins=5, density=True)
scores = -np.log(hist)

anomalies = [data[i] for i in range(len(data)) if scores[i] > 0.5]
print("异常值：", anomalies)
```

9.2.4 累积和法

累积和法（CUSUM）则常用于时间序列的异常检测，该方法通过监控偏差的累积和来检测数据中的持续偏移。结合我自己的研究方向，我对这一方法进行了重点关注，因为我自己的研究方向就是电力负荷预测，而在电力负荷预测研究中所使用的数据集通常就是时间序列数据，并且负荷偏移也是电力负荷中关键的数据特征，书中代码示例如下所示，在将来的负荷预测研究中对这些代码一定会对我自己研究产生很大的帮助。

```
# 示例代码：累积和法检测异常
import numpy as np

data = np.array([1, 2, 3, 4, 5, 100])  # 示例数据
mean = np.mean(data)
std_dev = np.std(data)

cumsum = np.cumsum(data - mean)
anomalies = np.where(np.abs(cumsum) > 3 * std_dev)[0]
print("异常点索引：", anomalies)
```

9.2.5 实践案例：基于箱线图的wiki网络流量异常检测

本节利用箱线图方法对wiki网络流量进行异常检测，书中还展示了如何使用箱线图识别不同语言页面流量的异常值。例如通过分析异常值，可以发现在热点新闻事件或节假日的影响下特定日期流量显著高于其他时间，书中示例代码如下，我也对其进行了相关运用，示例代码如下所示，程序运行结果在前文9.2.2节中已经进行了运用和展示。

```
import matplotlib.pyplot as plt
import pandas as pd

# 读取数据集
data = pd.read_csv("web_networktraffic.csv").dropna(axis=0, how='any')
data['Country'] = data['Page'].apply(lambda x: x.split('-')[1].split(")")[0])

# 数据预处理
country_list = ['en', 'ja', 'de', 'fr', 'zh', 'ru', 'es']
data = data[data['Country'].isin(country_list)]

plot_data = [[] for _ in range(len(country_list))]
for c, country in enumerate(country_list):
    plot_data[c] = [data.iloc[i, 2] for i in range(len(data)) if country ==
data.iloc[i, -1]]
    plot_data[c] = [data for data in plot_data[c] if data < 500][:8]

# 构建箱线图
plt.boxplot(plot_data, notch=True, vert=True, patch_artist=True)

# 可视化
colors = ['lightpink', 'lightblue', 'darkseagreen', 'wheat', 'thistle',
'rosybrown', 'lightgray']
for patch, color in zip(plt.boxplot(plot_data)['boxes'], colors):
    patch.set_facecolor(color)

plt.grid(axis='y')
```

```
plt.xticks(range(1, 8), labels=["英语", "日语", "德语", "法语", "中文", "俄语", "葡萄
牙语"], fontsize=16)
plt.yticks(fontsize=16)
plt.xlabel("语言", fontsize=16)
plt.ylabel("网络流量/字节", fontsize=16)
plt.tight_layout()
plt.show()
```

9.3 基于空间分布的异常检测

9.3.1 孤立森林

在进入到9.3节的学习之后，这一章节的第一小节为我介绍了一种基于空间分布的异常检测算法——孤立森林，我在之前的本科华中农业大学的python课程了解过；孤立森林（Isolation Forest，IF）是一种基于树结构的无监督异常检测算法，该算法通过随机选择特征和划分值来孤立数据点，从而判断异常值。

孤立森林由多个孤立树（Isolation Tree，iTree）组成。每个孤立树通过随机选择一个特征和一个划分值来划分数据集，直到所有数据点被孤立。孤立树的生成过程如下：

1. 随机抽取部分数据点。
2. 从特征空间中随机选择一个特征和一个划分值。
3. 将数据点与划分值进行比较，分为左右两部分。
4. 重复上述步骤，直到所有数据点被孤立或达到最大划分次数。

孤立森林通过计算数据点在孤立树中的路径长度来评估异常程度。路径越短，表示数据点越容易被孤立，越可能是异常点，书中这段对孤立森林算法的详细解释让我受益良多。

书中还介绍了孤立森林算法的程序设计步骤，如下所示：

1. 确定参数：孤立树的数量、最大划分次数、每次抽取的数据点数量和特征数。
2. 构建孤立森林，生成多个孤立树。
3. 计算路径长度：对于每个数据点，计算其在各孤立树中的路径长度。
4. 计算异常得分：根据路径长度计算数据点的异常得分。
5. 判断异常：设定阈值，若异常得分超过阈值，则标记为异常。

我在CSDN上寻找了一些相关案例和代码进行简单运用后，我对其有了更为深入的了解，正如书中所说的一样，孤立森林算法适用于大规模分布式系统，能够处理高维数据集；但是在处理高维数据集时存在局限性，因为其可能无法充分利用所有特征的信息，导致检测效果降低。此外，高维空间中的噪声特征或无关特征可能干扰树的构建过程，影响算法的准确性，这一点在CSDN上的许多推文也有所介绍。

9.3.2 局部异常因子（LOF）

局部异常因子（Local Outlier Factor，LOF）是一种基于数据局部密度的异常检测算法。它通过计算数据点与其邻域点的相对密度来判断异常程度。

书中所介绍的算法基础如下所示：

1. **k近邻距离**：数据点p到其第k个最近邻点的距离。
2. **k距离邻域**：在k近邻距离范围内的数据点集合。
3. **可达距离**：点p到点o的可达距离为点p的k近邻距离和点p与点o的直接距离中的较大值。
4. **局部可达密度**：点p与其k近邻的平均可达距离的倒数。
5. **局部异常因子**：数据点p的局部异常因子为局部可达密度的比值。

LOF算法通过计算点p与其k个最近邻点的相对密度来刻画其异常程度，这种相对密度计算方式有助于克服绝对密度估计的不足；通俗来说，即使在数据集的空间分布不均匀、密度差异明显的情况下，LOF算法也能取得良好的异常检测效果。

在充分阅读书中内容以及自己进行相关尝试之后，我总结出了LOF的程序设计步骤：

1. 数据去重：对数据集进行去重，避免重复样本对计算的影响。
2. 计算距离：分别计算每个数据点到样本空间内其余点的距离。
3. 排序：将距离升序排列。
4. 确定距离邻域：指定近邻样本个数，寻找每个数据点的距离邻域。
5. 计算局部可达密度：根据公式计算数据点的局部可达密度。
6. 计算异常得分：根据公式计算数据点的异常得分。
7. 判断异常：设定阈值，若异常得分超过阈值，则标记为异常。

我在同样对该算法进行了深入理解与初步应用，发现了LOF算法在空间分布不均匀的数据集中可以很容易的识别到异常数据点，但是该算法在处理大规模数据集时计算复杂度较高导致其效率相对低下。

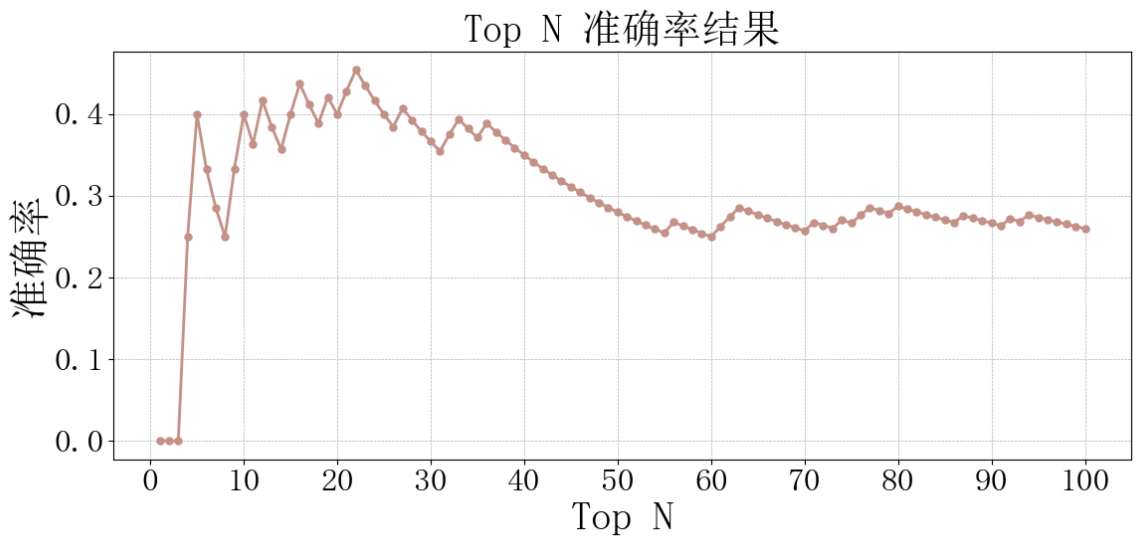
9.3.3 实践案例：基于局部异常因子的信用卡欺诈行为检测

书中本小节介绍了一个基于局部异常因子的信用卡欺诈行为检测的时间案例，该案例使用LOF算法对信用卡交易记录进行异常检测。数据集来源于Kaggle，包含284807笔交易记录，其中492笔为欺诈交易，书中示例代码如下所示：

```
# 文件名称：9-2LOF.py
# 说明：信用卡欺诈行为异常检测

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.neighbors import LocalOutlierFactor

# 数据读取
data = pd.read_csv("../data/creditcard.csv")
data = data.sample(n=50080, random_state=20)

# 特征工程
data["Hour"] = data["Time"].apply(lambda x: divmod(x, 3600)[0])  # 将秒转换为小时
X = data.drop(["Time", "Class"], axis=1)  # 特征矩阵
Y = data["Class"]  # 异常标签

# LOF异常检测
LOF = LocalOutlierFactor(n_neighbors=25)
pred = LOF.fit_predict(X)  # 训练并预测
Y_scores = LOF.negative_outlier_factor_  # 提取异常得分
data['scores'] = Y_scores

print("检测出的异常值数量为：", np.sum(pred == -1))

# 评估LOF算法性能
accuracies = []
for n in range(1, 101):
    df_n = data.sort_values(by='scores', ascending=True).head(n)
    accuracy = df_n[df_n['Class'] == 1].shape[0] / n
```

```
    accuracies.append(accuracy)

plt.figure(figsize=(12, 6))
plt.plot(range(1, 101), accuracies, color='#C69287', linestyle="-", linewidth=2,
marker='o', markersize=5)
plt.xlabel('TopN', fontsize=30)
plt.ylabel('准确率', fontsize=30)
plt.title("TopN准确率结果")
plt.xticks(np.arange(0, 101, step=10))
plt.yticks(np.arange(0, 1.1, step=0.1))
plt.grid(True, which="both", linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()
```

在对代码进行深入理解后，我对该代码进行了相关应用，得到了如下结果：



9.4 基于降维的异常检测

9.4.1 主成分分析（PCA）

主成分分析（Principal Component Analysis，PCA）是一种流行的数据降维技术，该技术通过识别数据中的主要变量和模式将高维数据有效地转换为较低维度的表示。在本书所讲的异常检测领域，PCA的应用主要集中在以下两个方面：

1. **基于降维的PCA异常检测**：通过将数据映射到低维空间，观察数据点在主成分上的投影，识别异常点。
2. **基于重构的PCA异常检测**：通过重构数据并计算重构误差，识别异常点。

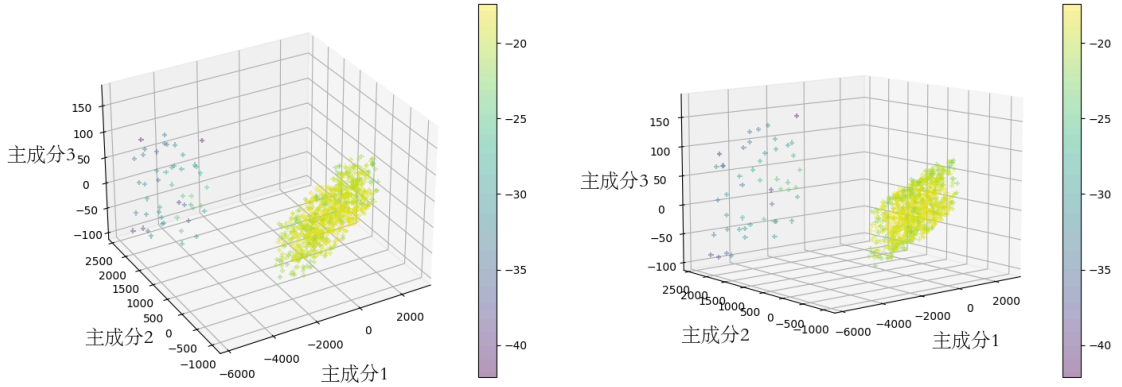
算法基础

1. **基于降维的PCA异常检测**：关注较小特征值对应的特征向量，计算数据点在这些方向上的偏离程度。
2. **基于重构的PCA异常检测**：将数据从低维空间重构回原始空间，计算重构误差，识别异常点。

算法步骤

1. 数据预处理：对数据进行标准化处理。
2. 特征值分解：计算各特征值及其对应的特征向量。
3. 降维：选择前k个特征值对应的特征向量作为主成分，得到降维后的数据。
4. 重构：将降维后的数据重构回原始空间。
5. 异常判定：根据重构误差计算异常得分，设定阈值，识别异常点。

在进行本章节学习之后，我了解到PCA在异常检测中具有诸多优势，如降低数据维度、消除噪声、无监督学习等。然而这个方法对数据中的异常值敏感，过多异常值可能会降低检测效果。此外，PCA基于线性变换，可能无法充分捕捉大规模数据中的复杂非线性关系，处于验证和进一步学习，我还对相关PCA代码进行运用，结果如下图所示：



9.4.2 自编码器

自编码器（Autoencoder，AE）也是一种降维技术，其通过学习数据的内部表示来重构输入从而捕捉数据的潜在结构；并且自编码器擅长区分数据的基本特征和噪声，因此常被用于检测异常。

自编码器由编码器和解码器组成，目标是最小化输入与重构输出之间的差异。模型训练的目标是最小化重构误差，通过反向传播算法优化参数。结合相关CSDN推文，我总结出其算法步骤为：

1. 数据预处理：对数据进行标准化处理。
2. 模型构建：构建自编码器，确定编码器和解码器的结构。
3. 模型训练：目标为最小化重构损失函数，通过反向传播算法优化参数。
4. 误差计算：计算每个数据点的重构误差。
5. 判断异常：设定阈值，若重构误差超过阈值，则视为异常。

9.5 基于预测的异常检测

9.5.1 向量自回归模型（VAR）

向量自回归模型（Vector Autoregressive model，VAR）是一种用于分析和预测多个时间序列之间相互关系的统计模型。它基于线性关系，利用所有当前时刻的变量值以及这些变量过去时刻的自身延迟值来构建。VAR模型的主要目的是研究一个系统中不同时间序列变量之间的动态影响，以及分析随机扰动对整个系统的冲击效应。

算法基础

假设有d维时间序列X，VAR模型的表达形式如下： $X_t=\mu+\Phi 1X_{t-1}+\Phi 2X_{t-2}+\cdots+\Phi pX_{t-p}+\epsilon_t$ 其中，X_t 是d维时间序列列向量，μ 是d维常数项列向量，Φi 是d×d阶矩阵，X_{t-i} 是滞后的时间序列，p是滞后阶数，ϵ_t 是d×1阶噪声向量。

VAR模型基于以下两个基本假设：

1. 多元时间序列的线性关系：VAR模型假设多个变量之间存在线性关系，并将每个变量表示为其他变量的线性组合。
2. 自回归结构：VAR模型使用自身的延退值作为解释变量，能够捕捉到变量之间的动态关系和时间序列的惯性。

算法步骤

1. 数据预处理：对数据进行标准化处理。
2. 平稳性检验：对各变量的时间序列数据进行平稳性检验（如单位根检验ADF），若均平稳，则可建立VAR模型。
3. 确定参数：使用信息准则（如AIC、BIC）或者交叉验证等方法来确定最优的滞后阶数p。
4. 构建模型：依据公式构建VAR模型。
5. 判断异常：计算每个时间步的预测误差error = $X_t-\hat{X}_t$，设定阈值s，若error > s，则标记时刻t为异常。

应用与评价

VAR模型能够捕捉多变量时间序列数据之间的动态关系，从而更好地识别异常。它能够考虑不同变量之间的相互影响，有助于发现异常事件对整体系统的影响，提高异常检测的准确性和可靠性。然而，VAR模型需要足够多的历史数据来建立有效的模型，这在某些情况下可能会限制其应用。此外，VAR模型在处理高维数据时计算复杂度较高，需要耗费较多的计算资源和时间。

9.5.2 自回归差分移动平均模型（ARIMA）

在书中本小节介绍了自回归差分移动平均模型（Autoregressive Integrated Moving Average，ARIMA），我之前在本科院校的数学建模课程上也进行过初步学习，这是一种经典的时间序列分析模型，常用于建模和预测具有自相关性和趋势性的时间序列数据。因此我也进行了重点关注和深入学习，结合相关资料，我了解到ARIMA模型由自回归（Autoregressive，AR）模型、差分（Integrated，I）模型和移动平均（Moving Average，MA）模型三部分构成。

1. **自回归模型（AR）**：关注时间序列中过去值对当前值的影响，通过构建滞后项与当前值之间的线性关系，提升数据中的自相关性。
2. **差分模型（I）**：用于处理非平稳时间序列，通过差分运算消除数据中的趋势和季节性因素，使序列变得平稳。
3. **移动平均模型（MA）**：考虑了预测误差的滞后影响，通过引入误差项的滞后项来平滑数据中的随机波动。

通过调整模型的参数，如自回归阶数p、差分阶数d和移动平均阶数q，ARIMA模型可以灵活适应不同类型的时间序列数据，捕捉其中的趋势、季节性和随机波动。

运用该模型进行程序设计的步骤为：

1. 数据预处理：对数据进行清洗和整理，处理缺失值、异常值和离群点，确保数据的质量和完整性。
2. 确定参数：指定参数p、d、q，得到初始化的ARIMA模型。

3. 模型训练：使用选定的ARIMA模型对历史数据进行训练。
4. 预测：使用训练好的ARIMA模型对数据进行预测。
5. 异常判定：计算每个时间步的预测误差，设定阈值，若误差大于阈值，则标记该时刻为异常。

在对该节内容进行深入学习后，我感受到了ARIMA模型在异常检测方面的独特的优势，它能够处理多种类型的时间序列数据，具有强大的适用性和灵活性；同时模型能够有效捕捉时间序列的趋势和季节性模式。但是ARIMA模型假设时间序列的生成过程是线性的，这大大限制了其在处理复杂非线性模式时的有效性；并且该模型性能高度依赖于参数选择，而合适的参数（p, d, q）往往难以确定，需要依赖于专业知识或试错法，这就导致了在大规模线性未知时间序列数据上，其表现可能不如LSTM等智能算法。

9.5.3 LSTM

长短时记忆人工神经网络（Long Short-Term Memory，LSTM）是一种特殊的循环网络（RNN），它能够有效地捕捉和处理时间序列数据中的长期依赖关系。其作为循环神经网络（Recurrent Neural Network，RNN）的演化变体，已经成为深度学习领域近年来最热门的算法之一。

要学习LSTM，首先来说一下循环神经网络，这是一类以序列数据为输入，在序列的演进方向进行递归且所有节点（循环单元）按链式连接的递归神经网络。其主要思想是将前一时刻的输出作为当前时刻的输入，并将当前时刻的输出传给下一时刻，以此来实现对序列数据的处理。这种时序的反馈机制使得RNN能够处理变长序列和捕捉序列中的动态模式。而与传统的循环神经网络相比，LSTM仍然是基于当前数据和上一时间点数据来计算下一时间点数据，只不过加入了输入门（input gate）、遗忘门（forget gate）和输出门（output gate）三个门，对内部的结构进行了更加精心的设计，能够更好地解决梯度爆炸和梯度消失问题。

该算法在我的本科许多课程中均有涉及，我也对其进行过不少应用，其算法步骤看似复杂，应用起来却十分简便易懂：

1. 数据预处理：对数据进行标准化处理。
2. 模型构建：构建LSTM模型，包括输入层、LSTM层和输出层。
3. 训练模型：目标为最小化损失函数Loss。
4. 判断异常：通常使用预测误差来判断异常。计算每个时间步的预测误差error = Xt−X^t，设定阈值s，若error > s，则标记时刻t为异常。

通过总结这段时间对LSTM模型的学习，我发现该模型在异常检测中的显著优势在于其能够有效地捕捉数据序列中的长期依赖关系，通过记忆单元的设计有助于学习和表征数据中的复杂模式，从而实现对异常行为的准确识别；但是随着数据维度和数据量的不断增加，该模型的训练对电脑配置和算力需求不断上升，结合个人应用经验，当数据量过大时，建议在云平台租用算力来解决这个问题。

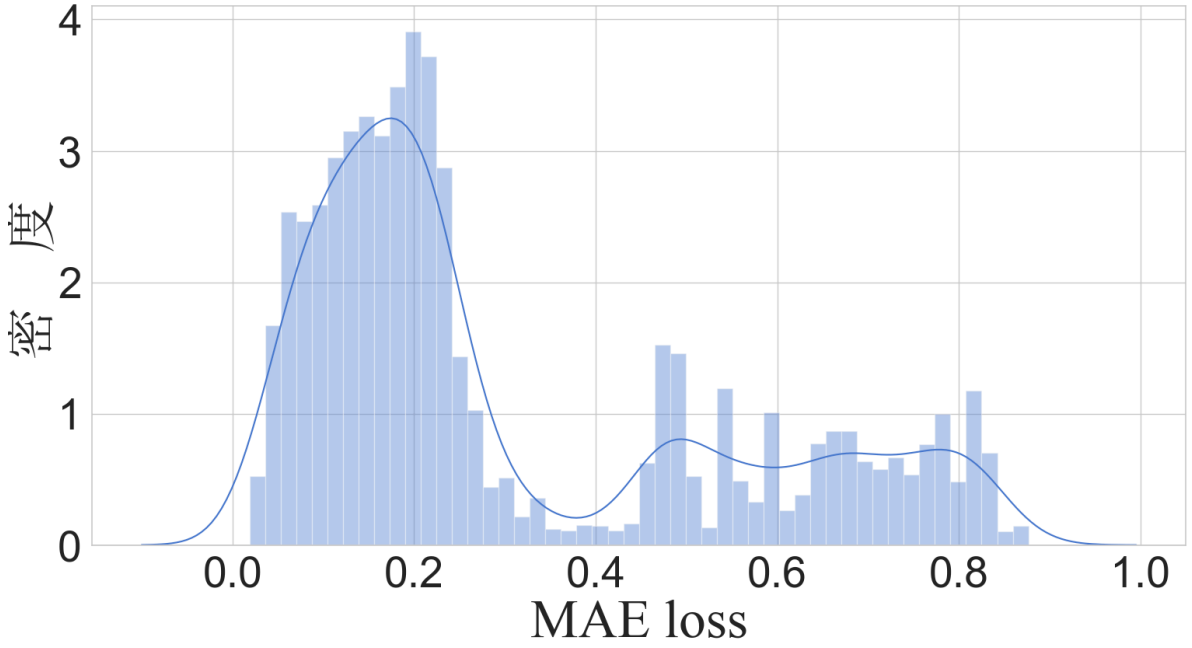
9.5.4 实践案例：基于LSTM的股票收盘价格异常检测

在本小节介绍的这个案例中，其使用LSTM算法对标普500指数（SPX）的日收盘价序列进行了异常检测。数据集的时间跨度为1986年1月2日至2018年6月29日，研究将训练集和测试集按照95:5的比例划分，测试集的时间跨度为2017年1月至2018年6月。该案例所使用代码如下所示：

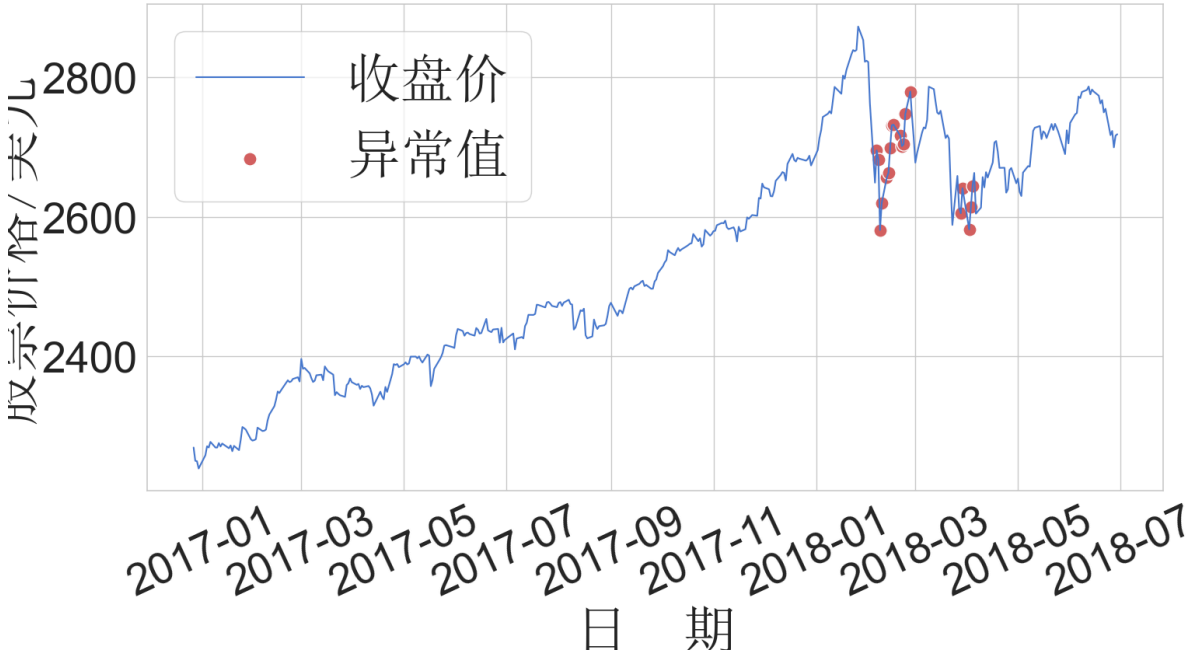
```
# 文件名称：9-4LSTM.py
# 说明：LSTM异常检测

import numpy as np
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import seaborn as sns
```

```
from pylab import rcParams
import matplotlib.pyplot as plt
from pandas.plotting import register_matplotlib_converters
from sklearn.preprocessing import StandardScaler

# 可视化设置
plt.rcParams['lines.linewidth'] = 6.0
plt.rcParams['font.sans-serif'] = ['SimSun']
plt.rcParams.update({'font.size': 24})
register_matplotlib_converters()
sns.set(style="whitegrid", palette="muted", font_scale=1.5)
rcParams['figure.figsize'] = 20, 1
```

在对该案例进行了充分理解后，我还对该案例进行了初步应用，得到的结果如下图所示：



这张图片展示LSTM模型在数据预测训练过程中的数据损失



这张图片则是展示了LSTM算法捕捉到这只股票的日收盘价序列中的数据异常点。

学习感悟

在学习了本书第九章——异常检测的内容后，我不仅深入掌握了书中介绍的一些异常检测算法的理论基础，还通过查找各种资料和实际代码应用加深了对这些算法的理解；尤其是像LSTM模型这样章节我还进行了重点学习，查找了许多相关的CSDN推文和期刊文献进行阅读学习和应用。我认为这些学习经历对我自己的研究方向——电力负荷预测研究非常有帮助，在未来的研究生学习生涯中，我也会不断增进自己的异常检测算法理解能力和代码应用能力，尽全力将书中内容应用到实际研究中并且推广到其他许多科研领域中去，感谢本书作者吕欣教授的倾囊相授。